VERIFICATION OF AN INDUSTRIAL SAFETY FUNCTION USING COLOURED PETRI NETS AND MODEL CHECKING

Tamás Bartha[1,2], András Vörös[2], Attila Jámbor[2], Dániel Darvas[2]

[1] *Computer and Automation Research Institute of the Hungarian Academy of Sciences (MTA SZTAKI)*
Kende u. 13–17., H-1111 Budapest, Hungary
bartha.tamas@sztaki.mta.hu

[2] *Department of Measurement and Information Systems, Budapest University of Technology and Economics (BME)*
Magyar tudósok körútja 2., H-1117 Budapest, Hungary
vori@mit.bme.hu

Abstract:

The verification of embedded, safety-critical industrial systems is important, since a failure of these systems may have catastrophic consequences. Formal methods guarantee not only the correctness, but also the completeness of the analysis. However, even moderately complex industrial systems have state spaces so large that former analysis techniques could not handle.

In this paper we model and analyse a small, but important part of a safety-critical industrial system: a safety function initiating an emergency procedure in a nuclear power plant. We model safety function using a proprietary coloured Petri net formalism, and perform the analysis by symbolic model checking based on the saturation algorithm. The analysis results were computed by the model checking tool developed at our department[*]*. Although this particular safety function has been analysed in earlier research [11], this is the first time the full behaviour of this system could be examined without any restrictions.*

Keywords:

safety systems, formal methods, coloured Petri net, model checking, saturation

1 INTRODUCTION

Embedded controllers are now a standard and prevalent part of industrial systems. They provide rich functionality and easy programmability. Still, these advantages also create a problem: the verification and validation (V&V) of these devices and their programs is becoming increasingly difficult. *Testing* is the traditional approach to V&V in industrial control systems. However, their behaviour is typically complex enough to make it impossible to achieve a complete test coverage for an even moderately complex controller. Hence, formal modelling and analysis is gaining wider acceptance in the industry, especially in the safety-critical application areas.

A frequently mentioned weakness of formal methods is that they often “bite off more than they can chew”, meaning that the formal models of real systems are susceptible to *state explosion*. While this is a valid argument, the aim of our paper is to demonstrate that recent development in the field of model checking, advanced state space exploration algorithms and storage data structures make us possible to solve problems that older methods could not handle. Our application example is a small, but important safety-critical industrial system: the safety function initiating an emergency procedure in a nuclear power plant.

The contributions of this paper are twofold: theoretical and practical. On the theoretical side, we have adapted and extended the so-called *saturation algorithm* [3] to be able to represent

[*] The Department of Measurement and Information Systems, BME.

and explore the state space of coloured Petri nets, and implemented this new approach in the model checking tool we are developing [14]. On the practical side, we have used this new algorithm to successfully verify the PRISE safety function (its detailed description is in Section 4.1) that had been analysed in earlier research [11], but its full behaviour could not be examined without restrictions by former approaches. As our measurements show, we could perform a complete analysis of this safety function quickly and efficiently, using only an ordinary desktop computer, by employing state-of the-art formal verification methods.

1.1 Previous work

As shown later in Section 5.2, our case study, the PRISE safety function has a huge state space ($> 10^{12}$ states) and many different behaviours, therefore efficient automatic methods are indispensable to prove its correctness. The first successful verification attempt was reported in [10], where the authors used coloured Petri nets and the Design/CPN modelling tool. Design/CPN has a simple explicit state model checker without built-in reduction methods, thus it was not able to explore the complete state space of the model, only a small part (approx. $4 \cdot 10^5$ states) could fit into the memory. The authors used state space reduction techniques, then partitioned the state space and separately analysed different subspaces. Finally, they have managed to obtain reduced subspaces with manageable size and could complete the formal verification.

Later, we have created a formal model of the PRISE safety function in the UPPAAL tool [12]. The modelling formalism of UPPAAL uses networks of timed automata extended with data structures and a data manipulation language. It has symbolic state space representation, built-in state space reduction methods, and a (partial) Computation Tree Logic (CTL) model checker. Unfortunately, UPPAAL has also failed to explore the complete state space due to memory overflow. Nevertheless, by reducing the model we have at least succeeded proving some of the requirements with UPPAAL.

We have also tried other symbolic approaches. Our first choice was the Symbolic Analysis Laboratory (SAL) model checker [12]. Sadly, this attempt to verify the PRISE safety function has failed as well, even though SAL uses a Binary Decision Diagram based efficient state space representation. Without being able to trace the low-level operation of SAL, our assumption is that the next-state relation grew too large: the state space explosion turned into decision diagram explosion in this case.

We have tried using other existing advanced Petri net verification methods [13]. They, however, operate on simple, uncoloured Petri nets, therefore we have developed an automated systematic conversion procedure to convert the coloured Petri net model of PRISE to a simple Petri net first.

- We have built an unfolding based analysis tool. As unfolding is efficient for asynchronous models, our expectation was that it could overcome the state space explosion problem. Unfortunately, this approach still ran out of memory due to the long distinctive trajectories.
- In [3] the authors showed an efficient symbolic state space generation and model checking method for asynchronous systems, especially for Petri nets. We have implemented and ran the algorithm with different settings on the converted simple net, but the algorithm ran out of memory. Unfortunately, the size of the converted model was too large, which caused both the state space representation and the next-state relation to exceed our resources.

The common weakness of the listed past approaches is that they could only reach partial success, as none of them was able to explore the full state space of the PRISE safety function.

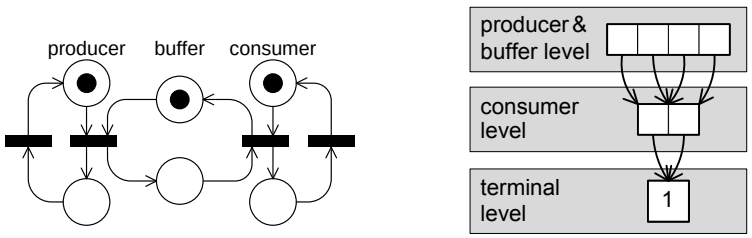
2 BACKGROUND

In this section we outline the theoretical background of our work. First, we present coloured Petri nets, the modelling formalism we used. Then, we introduce Multiple-valued Decision Diagrams. They form the underlying data structures of our algorithms that store the state space during model checking. Finally, we outline the saturation based state space exploration algorithm, and the model checking background.

2.1 Petri nets

Petri nets are graphical models for concurrent and asynchronous systems, providing both structural and dynamical analysis. A (marked) discrete ordinary Petri net is defined by a 5-tuple $PN = (P, T, E, w, M_0)$, represented graphically by a directed bigraph. $P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places, $T = \{t_1, t_2, \ldots, t_m\}$ is a finite set of transitions, $E \subseteq (P \times T) \cup (T \times P)$ is the finite set of edges, $w : E \to \mathbb{Z}^+$ is the weight function assigning weights $w(p_i, t_j)$ to the edges from p_i to t_j. $M : P \to \mathbb{N}$ is a marking function, where the number of tokens in a place p_i is represented by $M(p_i)$ for every i and M_0 is the initial marking of the net. A t transition is enabled, if for every $e = (p_i, t)$ incoming arc of $t : M(p_i) \geq w(p_i, t)$. An *event* in the system is the firing of an enabled transition t_i, which decreases the number of tokens in the input places p_j with $w(p_j, t_i)$ and increases the number of tokens in every p_k output places with $w(t_i, p_k)$. The firing of conflicting transitions is non-deterministic.

The *state space* or *reachability graph* of a Petri net is the set of states reachable from the initial state through transition firings. It can be either finite or infinite. Figure 1(a) depicts a simple example Petri net model of a producer-consumer system. The producer creates items and places them in the buffer, from where the consumer consumes them. For synchronizing purposes the capacity of the buffer is one, so the producer has to wait till the consumer takes the item from the buffer. This Petri net model has a finite state space of 8 states.



(a) The Petri net of the producer-consumer model

(b) State space representation with MDD

Figure 1: Producer-consumer example

2.2 Coloured Petri nets

The *coloured Petri net* (CPN) formalism enriches ordinary Petri nets with complex data structures [8]. This allows the creation of clearer and more compact models. In this paper we use a variant of well-formed coloured Petri nets that we built into the PetriDotNet tool [14], the software we develop and use for modelling and analysis.

Our net variant has a $CPN = (P, T, E, \Sigma, C, G, A, M_0^c)$ formal structure. The meaning of P, T and E is the same as in ordinary Petri nets. $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_\kappa\}$ is a set of colour sets (data types). In well-formed coloured Petri nets, the Σ set is a finite set. $C : P \to \Sigma$ is the colour function assigning colour sets to each place. $G : T \to \hat{G}$ is a function that assigns a guard to each transition. $A : E \to \hat{A}$ is the arc expression function assigning an arc expression to each edge. $M_0^c : P \to \hat{M}$ is a marking function assigning multi-sets of tokens to each place.

The firing semantic is different from ordinary Petri nets. A $G(t)$ guard is a logical function that can contain Boolean operators and place marking expressions. An $A(e)$ arc expression is a function that evaluates to a multi-set of tokens. The σ_i colour sets determine the allowed sets of tokens. A t transition is enabled, if for every incoming e arc the $A(e)$ expression is satisfiable and the value of the $G(t)$ guard is true. A firing of an enabled transition t_i takes $A(e)$ tokens from p for every $e = (p, t_i) \in E$ and puts $A(f)$ tokens to p for every $f = (t_i, p) \in E$.

The different variants of CPNs have various constraints for colour sets, guards and arc expressions. In our formalism the colour sets can be simple or complex colour sets. A simple colour set is a finite enumeration or a finite subset of integers. A complex colour set is a Cartesian product of simple colour sets. An arc expression can contain token constants and simple variables representing a member of a simple colour set. The guard expressions can contain token constants, simple variables, Boolean operators, relation signs and the successor operator [14].

2.3 Model checking

Model checking [5] is an automatic technique for verifying finite state systems. Given a model, model checking decides whether the model fulfils the specification. Formally: let M be a Kripke structure (i. e., state transition graph). Let f be a formula of temporal logic (i. e., the specification). The goal of model checking is to find all states s of M such that $M, s \vDash f$. *Structural model checking* approaches compute the results with the help of the formerly explored state space representation and transition relation representation. So at first the algorithm explores the possible reachable states, and after it we can perform the model checking procedure.

Computation Tree Logic (CTL) [6] is widely used to express temporal specifications of systems, as it has expressive syntax and there are efficient algorithms for its analysis. Operators occur in pairs in CTL: the path quantifier, either A (on all paths) or E (there exists a path), is followed by the tense operator, one of X (next), F (future, or finally), G (globally), and U (until). However we only need to implement 3 of the 8 possible pairings due to the duality [5]: EX, EU, EG, and we can express the remaining with the help of them in the following way: AX $p \equiv \neg$EX $\neg p$, AG $p \equiv \neg$EF $\neg p$, AF $p \equiv \neg$EG $\neg p$, A$[p$ U $q] \equiv \neg$E$[\neg q$ U $(\neg p \wedge \neg q)] \wedge \neg$EG $\neg q$, EF $p \equiv$ E$[true$ U $p]$.

The semantics of the 3 implemented CTL operators are as follows [5]:

- **EX**: $i^0 \vDash$ EX p iff $\exists i^1 \in \mathcal{N}(i^0)$ state so that $i^1 \vDash p$. This means that EX corresponds to the inverse $\mathcal{N}$ function, applying one step backward through the next-state relation.
- **EG**: $i^0 \vDash$ EG p iff $\forall n > 0 : \exists I = (i^0, i^1, i^2, \ldots, i^n)$ path, so that $\forall 1 \leq j \leq n : i^j \in \mathcal{N}(i^{j-1})$, and $i^j \vDash p$, so that there is a strongly connected component containing states satisfying p. The evaluation of EG needs a greatest fixed-point computation, therefore saturation cannot be applied directly to it. Computing the fixed-point, however, benefits from the locality provided by decomposition.
- **EU**: $i^0 \vDash$ E$[p$ U $q]$ iff $\exists n \geq 0, \exists I = (i^0, i^1, i^2, \ldots, i^n)$ path, so that $\forall 1 \leq j \leq n : i^j \in \mathcal{N}(i^{j-1})$, $\forall 0 \leq k < n : i^k \vDash p$ and $i^n \vDash q$. The sates satisfying this property are computed with the following least fixed-point: **lfp** $Z[q \vee (p \wedge$ EX $Z)]$. Informally: we search for a state q reached through only states satisfying p.

2.4 Decision diagrams

Decision diagrams [1] are used in symbolic model checking for efficiently storing the state space and the possible state changes of the models. A *Multiple-valued Decision Diagram* (MDD) is a directed acyclic graph, representing a function f consisting of K variables: $f : \{0, 1, \ldots\}^K \to \{0, 1\}$. An MDD has a node set containing two types of nodes: non-terminal and two terminal nodes (0 and 1). The nodes are ordered into $K + 1$ levels. A non-terminal node is labelled by a variable index $1 \leq k \leq K$, which indicates to which level the node belongs

(which variable it represents), and has n_k (domain size of the variable) arcs pointing to nodes in level $k-1$. A terminal node is labelled by the variable index 0. Duplicate nodes are not allowed, so if two nodes have identical successors in the lower level, they are also identical. These rules ensure that MDDs are canonical and compact representation of a given function or set. The evaluation of the function is the top-down traversal of the MDD through the variable assignments represented by the arcs between nodes. Figure 1(b) depicts an MDD used for storing the encoded state space of the example Petri net. Each edge encodes a possible local state, and the possible global states are the paths from the root node to the terminal *one* node.

2.5 Saturation

Saturation is a *symbolic algorithm* for state space generation and model checking. *Decomposition* serves as the prerequisite for the symbolic encoding: the algorithm maps the state variables of the chosen high-level formalism into symbolic variables of the decision diagram. The global state of the model can be represented as the composition of the local states of components: $s_G = (s_1, s_2, \ldots, s_n)$, where n is the number of components. In addition, decomposition helps to efficiently exploit locality, which is inherent in asynchronous systems. Locality ensures that a transition usually affects only some components or some parts of the sub-models. The algorithm does not create a large, monolithic next-state function representation. Instead it divides the global next-state function into smaller parts, according to the high-level model. Formally: $\mathcal{N} = \bigcup_{\forall e \in \mathcal{E}} \mathcal{N}_e$, where $\mathcal{E}$ is the set of events in the high-level model. The granularity of the decomposition, i. e., the next-state relations represented by $\mathcal{N}_e$ can be chosen arbitrarily.

Saturation uses *symbolic encoding of the next-state function*. We used the symbolic next-state representation from [2, 4]. This approach partitions disjunctively the global next-state function according to the high-level model events in the system. Logically, if $\mathcal{N}$ is represented by the relation between state variables (in the decision diagram representation) $\vec{x}, \vec{x}'$ with $\mathcal{R}_e(\vec{x}, \vec{x}')$, then the global relation can be expressed by the symbolic next-state relations of the events: $\mathcal{R}(\vec{x}, \vec{x}') = \bigvee_{\forall e \in \mathcal{E}} \mathcal{R}_e(\vec{x}, \vec{x}')$. This way the algorithm can use smaller next-state representations.

However, in many cases the computation of the local $\mathcal{N}_e$ functions are still expensive. The algorithm handles this problem by conjunctive partitioning according to the enabling and updating functions [4]: $\mathcal{N}_e = \bigcap_{\forall i: 0 \le i \le n} \left(\mathcal{N}_{e,i}^{enable} \bigcap \mathcal{N}_{e,i}^{update} \right)$, which can be symbolically computed by $\mathcal{R}_e(\vec{x}, \vec{x}') = \bigwedge_{\forall i: 0 \le i \le n} \left(\mathcal{R}_{e,i}^{enable}(\vec{x}, \vec{x}') \bigwedge \mathcal{R}_{e,i}^{update}(\vec{x}, \vec{x}') \right)$. Applying $\mathcal{N}_e$ to a given set of states represented by *states* results $\mathcal{N}_e(\textit{states}) = \textit{RelProd}(\mathcal{R}_e(\vec{x}, \vec{x}'), \textit{states})$, where *RelProd* is the well-known relational product function [5]. The smaller the partitions we create, the less computation they need. The limit for the size of the partitioning comes from the used high-level modelling formalism.

Saturation uses a *special iteration strategy*, which is efficient for asynchronous systems [3]. Building the MDD representation of the state space starts at the MDD of the initial state. Then the algorithm saturates every node in a bottom-up manner, by applying saturation recursively, if new states are discovered. In this way saturation avoids the peak size of the MDD to be much larger than the final size, which is a critical problem in traditional approaches.

3 SATURATION ALGORITHM FOR COLOURED PETRI NETS

As shown in Section 1.1, existing low-level models and inefficient model checking algorithms prevented us from reaching our goal: to fully verify the PRISE safety function. Therefore we selected coloured Petri nets as the modelling formalism, and saturation as the basis of state space exploration and traversal. However, saturation has not supported CPNs at that time, thus we needed to develop it further. In this section we introduce our approach to saturation-based state space discovery for coloured Petri nets.

3.1 Next-state function representaton

There are several important differences between simple and coloured Petri nets that required us to modify and extend both the MDD-based state representation and the saturation algorithm. The first fundamental difference is that the decompositions of coloured Petri nets are not *Kronecker consistent* [9], due to the additional net elements (e. g., guard expressions). Therefore, the global next-state function of a decomposed CPN model cannot be computed as the intersection of the local next-state functions of the sub-models, i. e., $\mathcal{N}_e = \bigcap_{\forall i \in e} \mathcal{N}_{e,i}$ does not hold. As a consequence, the so called *Kronecker matrices* cannot be applied for storing the next-state function $\mathcal{N}$. Our approach uses an MDD-based representation of the state transition relations instead.

Assume we decompose the coloured Petri net to K sub-models, just as we did in Section 2.5. The set of possible states of the k-th sub-model is $\mathcal{S}_k$. The set $\mathcal{L}_k$ denotes the states directly reachable from $\mathcal{S}_k$ by a single firing. Applying conjunctive partitioning, we can describe the $\mathcal{N}$ next-state function by a $2K$-level MDD denoted as $\mathcal{R}$. The $i_k : 1 \leq k \leq K$ denotes the state at the k-th level of the MDD we are in *before* firing the e event (i. e., $\mathcal{R}_e^{enable}$), while $j_k : 1 \leq k \leq K$ denotes the state at the k-th level we will get to *after* firing the e event (i. e., $\mathcal{R}_e^{update}$). Consequently, $i_k, j_k \in \mathcal{S}_k$ and $1 \leq k \leq K$. Therefore, a $(i_K, j_K \ldots, i_1, j_1)$ sequence represents the global state transition, where the initial state is $(i_K \ldots, i_1)$, and the state after the firing is $(j_K \ldots, j_1)$. A $(i_K, j_K \ldots, i_1, j_1)$ transition is enabled and fireable iff the corresponding $(i_K, j_K \ldots, i_1, j_1)$ directed path leads from the root of $\mathcal{R}$ to the terminal 1 node.

3.2 Event handling

Coloured Petri nets can model complex systems in a very compact form by utilizing the data content of tokens instead of pure structural constructs. However, this compactness takes its toll during state traversal: the local state spaces of the sub-models in a decomposed CPN are typically much larger and more complex than in simple Petri nets. Moreover, in CPNs less variables are used to encode the same set of states into decision diagrams, thus there is less redundancy in the state space representation, resulting in a less efficient form of storage. Our previous research proved that the smaller the partitions are, the more efficient saturation becomes, since the creation and maintenance of the smaller parts requires significantly less resources. The aim of the *conjunctive refinement* of the partitioning, as described in this section, is to further decompose the state transitions into smaller parts, and to treat these parts separately and efficiently. The steps of this *event handling* process are shown in Figure 2.

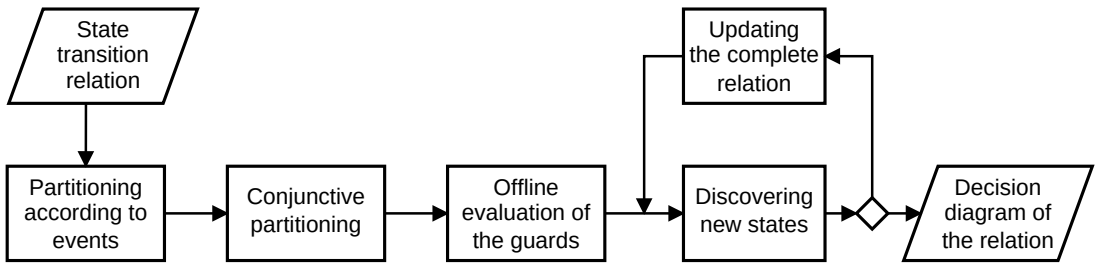


Figure 2: Flow diagram of the event handling

1) *Partitioning according to events*: the state transition relation is partitioned in accordance with the events. The $\mathcal{N}_e$ relations of the $e \in \mathcal{E}$ events are stored in separate $\mathcal{R}_e$ MDDs. The original state transition relation can be calculated as $\mathcal{R} = \bigvee_{\forall e \in \mathcal{E}} \mathcal{R}_e$.
2) *Conjunctive refinement of the partitioning*: the above partitioning is further refined by splitting the $\mathcal{N}_e$ state transition relation of each e event into K parts, and encoding them into

$\mathcal{R}_{e,k}$ MDDs (where $1 \leq k \leq K$). In order to deal with Kronecker inconsistency, the fireable bindings of the variables are also stored within the MDDs. Furthermore, another MDD is created to represent the constraints imposed by the guards associated with the events. This MDD, denoted as $\mathcal{R}_e^{guard}$, stores those bindings of the variables in the input and output arc expressions of the transition for which the guard evaluates to true.

3) *Off-line evaluation of the guards*: The $\mathcal{R}_e^{guard}$ MDD that represents the constraints imposed by the guard associated to the e event is created before the iteration of the saturation. The variable bindings stored in this MDD need not be updated later during saturation.
4) *Discovering new states*: As soon as a new state is discovered during the iteration of the saturation, the fireable state transitions from this new state must be instantiated immediately. When a new state transition is found on the k-th level as a result of firing the e event, then the $\mathcal{R}_{e,k}$ MDD is expanded with this new state transition, and in addition to it the variable binding that makes the state transition fireable is also stored.
5) *Updating the complete relation*: Since the original iteration order of the saturation is preserved in our algorithm, the complete state transition relation must be recreated by intersecting the MDDs of the partial relations: $\mathcal{R}_e = \left(\bigwedge_{\forall k:1\leq k\leq K} \mathcal{R}_{e,k}\right) \wedge \mathcal{R}_e^{guard}$.

3.2.1 Off-line evaluation of the guards

During the initialization of the saturation algorithm the $\mathcal{R}_e^{guard}$ MDDs storing the constraints by the guards of each e event need to be created. These MDDs are constructed in four steps:

1) The variables included in the guard expression are collected.
2) A new level is built for each variable in the MDD. These levels are inserted above the levels corresponding to the state variables.
3) The variables are bound for every combination of values permitted by their colour sets in an exhaustive manner.
4) For each possible binding the guard is evaluated. Every binding that evaluates to *true* is stored in the decision diagram, since with this binding the guard permits the firing of the transition. For this purpose each colour in the colour set of a token variable in a newly created level is associated with an integer. This way a binding stored in the decision diagram can be mapped to a directed path.

An $\mathcal{R}_e^{guard}$ MDD initialized with the above steps contains the possible bindings that make the guard enable the firing of the transition. Since the guard expression does not change during the execution of the model, it is not necessary to update the conjunct represented by the MDD during saturation.

3.3 Other modifications

We needed to adapt the `Saturate()` and `SatFire()` methods of the classical saturation algorithm [3] to the extended data structures of the state transition relation. As the decompositions of the coloured Petri nets are not guaranteed to be Kronecker consistent, the set of enabled transitions on the k-th level may depend on the already visited states on higher levels. Consequently, we modified the `Saturate()` and `SatFire()` functions so that during the depth-first traversal they move downward in the $\mathcal{R}_e$ relations consistent with the fired transitions, and the resulting MDD nodes are always passed to the functions called on deeper levels of the recursion. The pseudo-code of the modified `ColouredSaturate()` and `ColouredSatFire()` functions is listed in Algorithm 1 and 2.

Algorithm 1 ColouredSaturate

Input: p : node

1: $k \leftarrow Level(p)$
2: $chng \leftarrow$ **true**
3: **while** $chng$ **do**
4: $chng \leftarrow$ **false**
5: **for all** $e : Top(e) = k$ **do**
6: **for all** $i \in \mathcal{S}_k,\ j \in \mathcal{L}_k :$ $p[i] \neq \mathbf{0} \wedge \mathcal{R}_e[i][j] \neq \mathbf{0}$ **do**
7: $f \leftarrow$ ColouredSatFire(e, $p[i]$,$\mathcal{R}_e[i][j]$)
8: **if** $f \neq \mathbf{0}$ **then**
9: $u \leftarrow$ Union(f, $p[j]$)
10: **if** $u \neq p[j]$ **then**
11: $p[j] \leftarrow u$
12: $chng \leftarrow$ **true**
13: **if** $j \notin \mathcal{S}_k$ **then**
14: ColouredConfirm(k,j)
15: **end if**
16: **end if**
17: **end if**
18: **end for**
19: **end for**
20: **end while**

Algorithm 2 ColouredSatFire

Input: e : event, p : node, $\mathcal{R}$: relation

1: $l \leftarrow Level(p)$
2: **if** $l < Bot(e)$ **then return** p
3: **if** CachedFire(e, p, out s) **then return** s
4: $s \leftarrow$ NewNode(l)
5: $chng \leftarrow$ **false**
6: **for all** $i \in \mathcal{S}_k,\ j \in \mathcal{L}_k : p[i] \neq \mathbf{0} \wedge \mathcal{R}[i][j] \neq \mathbf{0}$ **do**
7: $f \leftarrow$ ColouredSatFire(e, $p[i]$, $\mathcal{R}[i][j]$)
8: **if** $f \neq \mathbf{0}$ **then**
9: $u \leftarrow$ Union(f, $p[j]$)
10: **if** $u \neq p[j]$ **then**
11: $p[j] \leftarrow u$
12: $chng \leftarrow$ **true**
13: **if** $j \notin \mathcal{S}_k$ **then**
14: ColouredConfirm(k,j)
15: **end if**
16: **end if**
17: **end if**
18: **end for**
19: **if** $chng$ = **true then** ColouredSaturate(s)
20: CheckIn(s)
21: PutInCacheFire(e, p, s)
22: **return** s

4 THE MODELLED INDUSTRIAL SYSTEM

The subject of our research is a safety function, designed to initiate an emergency prevention action in the occurrence of the so-called *PRISE event*. This safety function is used in the Paks Nuclear Power Plant (Paks NPP) located in Hungary. The Paks NPP operates four VVER-440/213 type pressurized water reactor (PWR) units with a total nominal (electrical) power of approx. 2 GW. Nuclear power plants are highly safety-critical and complex systems, where the correct operation of the safety procedures is of great importance. The plant protection systems must satisfy high safety requirements and minimize spurious forced outages. Therefore, formal modelling and verification methods need to be applied to prove the correctness and completeness of the PRISE safety function.

The *PRImary-to-SEcondary leaking* (PRISE) event is one of the major faults in a reactor unit, resulting due to a non-compensable leaking of parts in the primary circuit. The PRISE event occurs when there is a rupture or other leakage within the steam generator (SG) vessel primary tubing, affecting either a few (3-10) tubes or their collector that contain the high-pressure activated liquid of the primary circuit. The PRISE event is the VVER-440/213 analogue of the well-investigated Steam Generator Tube Rupture (SGTR) event (see e. g., [7]) in other types of pressurized water reactors.

In the unlikely case of a PRISE event, the safety procedures first initiate the emergency shutdown (scram, trip) of the reactor, and then isolate the faulty steam generator. However, there would still be a possibility to release some of the contaminated water to the environment, if the event would not be handled properly. In order to prevent this and to increase the safety of the plant, a safety valve for draining the contaminated water into the containment has been added to each steam generator, and a new safety function, the *PRISE safety function* has been developed to control its operation.

4.1 The PRISE safety function

The technological and I&C system experts of the Paks NPP have designed a timed logical scheme, the basis of the PRISE *safety function*, in a heuristic way. The logical scheme was specified as a Functional Block Diagram (FBD) representation (a formalism similar to the one defined in the IEC 61131-3 standard). The PRISE safety function FBD is shown in Figure 3. The description of the inputs and outputs of the PRISE safety function are included in Table 1.

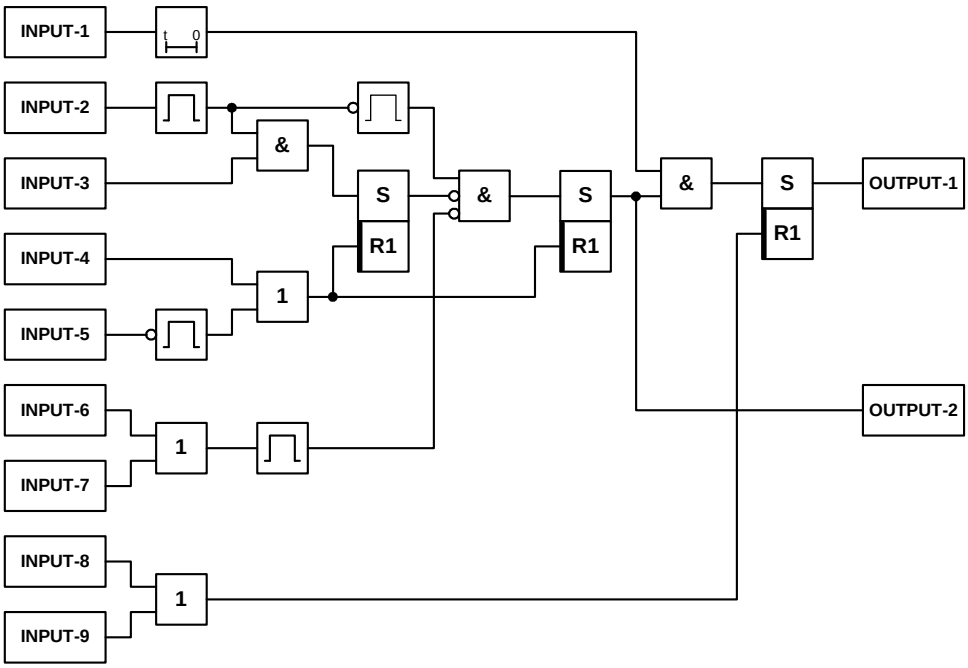


Figure 3: Functional block diagram of the PRISE safety function [11]

The purpose of the PRISE safety function is to initiate the draining of the steam generator *if and only if a PRISE event occurs*. This implies preventing the activation of the safety valve, when a non-PRISE fault event (causing similar symptoms, but without a classified PRISE event) occurs, i. e., the PRISE safety function must be *selective*. Moreover, when the reactor unit is either being started up or shut down, thus it is not in the normal operating regime, the PRISE safety function is designed not to be active. In these circumstances the operators can activate the draining valve manually, should a need arise.

The designed safety procedure initiates the draining (OUTPUT-1) when a critical decrease in the primary pressure (INPUT-2) is followed (after a specified time delay) by the increase of the steam generator level (INPUT-1) that lasts for a certain time interval. However, the draining is initiated only if the containment pressure keeps its nominal value (INPUT-3), i. e., it is not increasing due to another, non-PRISE fault causing an inflow of the primary water into the containment. The minimum time interval constraint for INPUT-1 to hold its value prevents the incorrect initiation of draining by an unreliable water level sensor measurement showing temporarily a spuriously high value (caused by the solid scale content of the secondary water).

The INPUT-4 and INPUT-9 input conditions inhibit the operation in a startup or shutdown situation. INPUT-5 resets the operation of the PRISE safety procedure in case the reactor is being shut down. INPUT-6 and INPUT-7 prevent the erroneous draining of the containment after the isolation of a steam generator caused by a non-PRISE fault. INPUT-8 indicates the situation when the steam generator was manually isolated due to a failure indication. The primary OUTPUT-1 of the procedure is the presence of a PRISE event. Note that the auxiliary OUTPUT-2 signal indicates the presence of all but one of the symptoms of the PRISE situation.

Table 1: PRISE safety procedure I/O description

Name	Description	Function
INPUT-1	SG level high	Steam generator water level is increasing (due to closure of the turbine)
INPUT-2	Primary pressure decreasing	The pressure of the primary water is decreasing (due to PRISE or other leakage)
INPUT-3	Containment pressure is normal	The pressure of the containment is *not* increasing (no primary water inflow caused by a non-PRISE fault)
INPUT-4	Primary temperature below nominal	Technical condition signifying that the reactor is in startup/ shutdown regime
INPUT-5	Control rods fully down	Technical condition used to reset the operation of the PRISE safety procedure
INPUT-6	SG deltaP	Technical conditions used to avoid the erroneous draining of
INPUT-7	SG RAP 1/2	the secondary water after isolation of the steam generator
INPUT-8	SG inhibition	Technical condition used to indicate the SG inhibited state
INPUT-9	Primary pressure low	Technical condition signifying that the reactor is in startup/ shutdown regime
OUTPUT-1	SG is inhermetical	Primary output, activates the secondary water drain
OUTPUT-2	ACTIVE	Auxiliary output used in control operations

4.2 Coloured Petri net model of the PRISE safety function

We have created a hierarchical Coloured Petri net model of the PRISE safety function. Figure 4 shows the high-level main net of our CPN model. The gray circles are the inputs and outputs of the PRISE logic. The larger labelled rectangles are substitution transitions that denote subnets of the corresponding function blocks. The smaller net elements are simple places and transitions that are only needed for connecting the subnets. This main net integrates and connects the separately developed and validated lower-level CPN subnets of the different functional blocks. The transformation of the Functional Block Diagram (see Figure 3) was straightforward and simple to validate, since the structure of the FBD graph and the corresponding CPN graph are isomorphic.

The run-time environment is a safety-critical, highly dependable digital distributed control system (DCS), which runs at an explicit 50 millisecond long *scan cycle*. During each scan cycle the controller first samples its inputs, then evaluates all of its functional diagram pages starting from the blocks connected to the inputs and following the flow of data until they reach the outputs, computes its new internal state, sets the outputs, and in the remaining time performs self-tests. This behaviour is reflected by the CPN model the following way: the propagation of the tokens in the net represents the flow of data in the functional diagram. The CPN model has a feedback loop that puts simultaneously a single coloured token into each input place at the beginning of a scan cycle. The colour of the input tokens carries the input data value. These tokens initiate the execution of the subnets modelling the function blocks. When every subnet has been executed, a single coloured token is generated into each output place. The feedback loop takes away every generated token from the outputs and the scan cycle ends.

An example CPN subnet —modelling the operation of a functional block, namely the *Delay module*— is shown in Figure 5(a). The functionality of the Delay module is given by a time diagram in Figure 5(b). The purpose of the module (as its name implies) is to delay a rising edge pulse for a predefined D number of cycles. When the module detects a rising edge, it

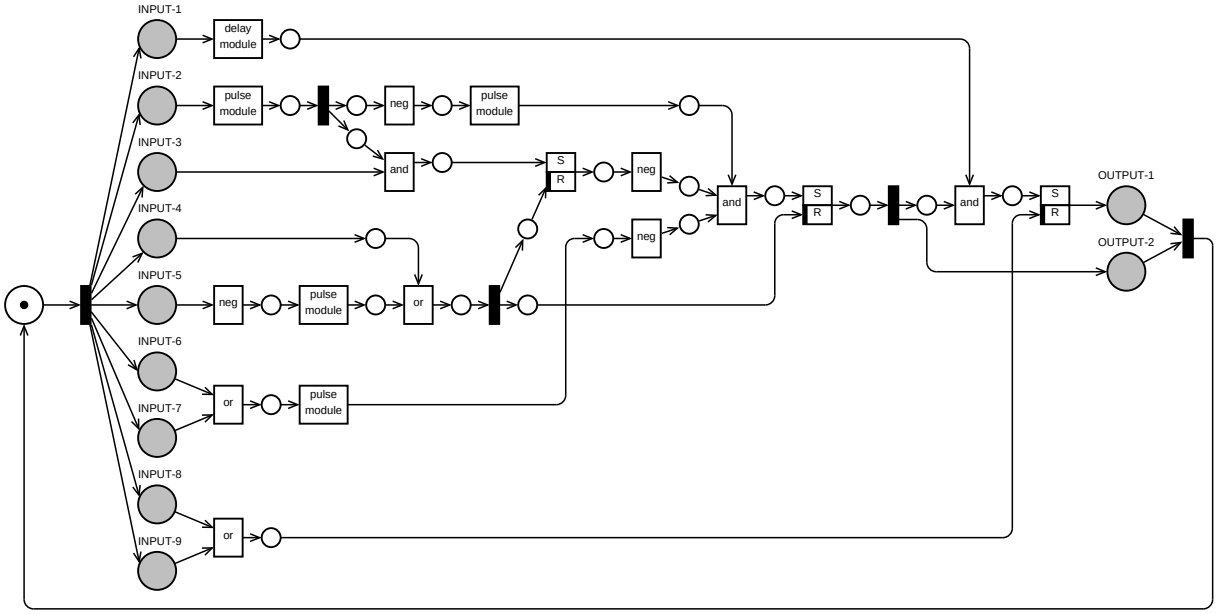
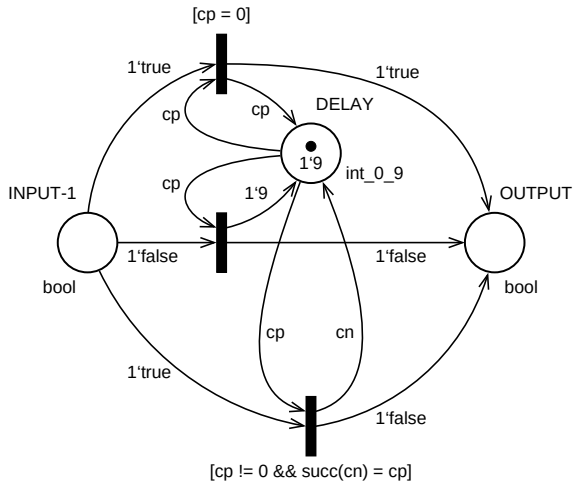
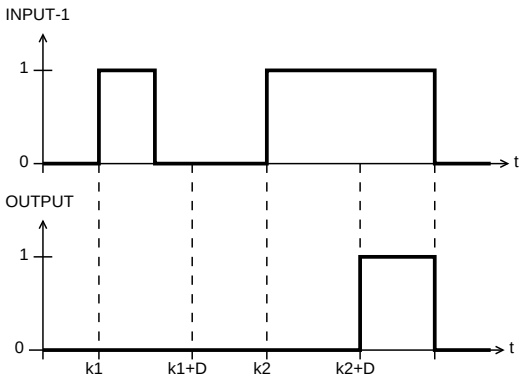


Figure 4: The Coloured Petri net model of the PRISE safety procedure

starts a counter. If the pulse is active (the input remains 1) for at least D number of cycles, the Delay module will "let the pulse pass", that is it sets its output to 1 (the `true` Boolean value). The output will remain 1 as long as the input is active. When a falling edge is detected, the module resets itself to its default inactive state.



(a) The CPN subnet model of the Delay module



(b) Time diagram of the operation

Figure 5: Delay module: model and operation

The operation of the CPN subnet model of the Delay module (see Figure 5(a)) is easy to follow. The model has two *port places* (the INPUT-1 port, and the OUTPUT port) that represent the connections of the Delay module. The DELAY place stores the value of the delay counter. Its colour set is `int_0_9`, a subset of integers: $\{0, \ldots, 9\} \subset \mathbb{Z}$. Its initial marking has one token whose colour equals the required delay time, given as D cycles (in the example $D = 9$). The three black rectangles are the transitions that realise the three main phases of the operation. The expressions written in brackets next to the transitions are their *guards*. A guard is a

boolean expression that prohibits the firing of the corresponding transition unless it evaluates to `true`. The three main phases of the operation and their transitions are as follows:

1) The *lower transition* detects the rising edge (a `true` value is in the input and the delay counter has not yet reached zero), starts the delay counter, and continues counting down in the subsequent cycles. The guard prescribes the previous value of the counter to be the successor of the next value, thus implementing the counting down process. The output remains inactive in this phase.
2) The *upper transition* will fire whenever the delay counter has ran out (reached zero), and the input is active. The transition puts a token with `true` value in the output port place, therefore the output will remain active as long as the input is active.
3) The *middle transition* detects the falling edge of the pulse, resets the value of the delay counter, and deactivates the output.

5 VERIFICATION OF THE PRISE SAFETY FUNCTION

Our aim was to prove that the PRISE safety function initiates the draining *always if a PRISE event occurs* in every normal operation regime coupled with a *fault in the SG level sensor* that is highly unreliable; and *never if a PRISE event does not occur* even if severe faults causing similar symptoms happen. In addition, it is also important to prove that the PRISE detection logic is free from deadlocks as they represent dangerous situations. The required selective detection of the PRISE event, and the heuristic design process of the safety logic made it necessary to perform a rigorous formal verification of the PRISE safety procedure.

5.1 Formalization of the requirements

We could translate the above requirements into the following verification goals:

- *Liveness requirement:* the secondary water draining activity is always activated when a real PRISE accident has occurred (no actuation masking).
- *Safety requirement:* the draining activity is not activated if not a real PRISE accident has occurred (no erroneous actuation).
- *Deadlock freeness:* No deadlock situation can arise for any combination and sequence of input signals.

We used branching-time temporal logic based model checking to prove the requirements. For complexity reasons we chose CTL temporal logic, as it provides an expressive formalism with efficient decision procedures.

- First, we checked the deadlock freeness of the system. Informally this means that in every state there exists at least one reachable successor state. The equivalent CTL temporal logic expression is the following: AG(EX(*true*)).
- We also checked if the model is *reversible*, that is from every state we can reach the initial state. We expressed it with the following CTL formula: AG(EF([*init*])). This property ensures that the safety function can be made ready to fulfil its purpose in all circumstances.
- We used indirect proof to prove the safety requirement. We transformed the inverse requirement into the following CTL formula: E($\neg$ [*PRISE-event*] U [*actuation*]). This formula is satisfied only if the draining activity is activated without a PRISE event.
- The liveness requirement was also easier to prove by indirect proof. We formalised the inverse requirement as the following CTL expression: EF([*PRISE-event*] $\wedge$ EG($\neg$ [*actuation*] $\wedge$ $\neg$ [*reset-event*])). Informally, we are searching for strongly connected components in the state space that contain no *actuation* and *reset-event*, but contain a *PRISE-event*.

5.2 Evaluation of the temporal expressions

The next step of the verification was to explore and store the state space of the CPN model of the PRISE safety function, using our coloured saturation algorithm and state space storage data structures described in Section 3. After obtaining the complete state space we could evaluate the four CTL expressions introduced in the previous section. For state space traversal and temporal logic based model checking we developed our own experimental implementation of our algorithms written in the C# programming language. We used the following configuration for our measurements: Intel L5420 2.5 GHz processor, 8 GB memory, Windows Server 2008 R2 (x64) operation system, .NET 4.0 runtime. The measurement results are listed in Table 2.

Table 2: Characteristics of the state space traversal

Parameter	**Value**
Run-time	950 s
Number of global states	$4.836 \cdot 10^{12}$
State space representation (nodes)	1 497
Number of local state changes	10 082 881
Transition representation (nodes)	782 159

Run-time represents the time needed to explore the state space. The state space generation required 950 s for the PRISE CPN model. The deadlock freedom and reversibility checking temporal expressions took 6 s each to evaluate on the existing state space representation. The liveness and safety requirements were evaluated in 2 s and 3 s, respectively.

Other characteristics of the state space shown in Table 2 include the *number of global states*, which is the size of the state space, i. e., the number of reachable states from the initial state. The *state space representation* gives the number of MDD nodes used to represent the state space symbolically. The *number of local state changes* means the number of symbolically enumerated possible state changes. *Transition representation* is the number of stored MDD nodes in the symbolic next-state representation.

As the measurements show, we could quickly and efficiently complete the earlier unachievable task (see Section 1.1) of fully verifying the PRISE safety function. However, state space generation is still challenging, even for state-of-the-art model checking tools. In our case, despite our efficient next-state representation and state traversal algorithm, building the next-state relation required an order of magnitude more nodes than the state space representation.

6 CONCLUSION

This paper presented the formal verification of a safety function initiating an emergency procedure in a safety-critical environment. The contributions of the paper are twofold:

- Although this safety function has been analysed in previous research, this is the first time its full state space could be explored without restrictions, using our own implementation and an ordinary desktop computer.
- We could achieve this result by extending the highly efficient saturation algorithm into the domain of coloured Petri nets. Since there are many non-trivial differences between simple and coloured Petri nets, the extension required a novel state space storage structure and appropriate modifications of the state space traversal algorithm.

Our future research is directed towards improving the performance of the algorithm by further partitioning the state transition representation.

REFERENCES

[1] **Bryant, R.E.** (1986): *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions on Computers, 35, pp. 677–691.

[2] **Burch, J.R.; Clarke, E.M.; Long, D.E.** (1991): *Symbolic Model Checking with Partitioned Transition Relations*. In: Proceedings of the International Conference on Very Large Scale Integration, pp. 49–58.

[3] **Ciardo, G.; Marmorstein, R.; Siminiceanu, R.** (2003): *Saturation Unbound*. In: Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 379–393, Springer, DOI: 10.1007/3-540-36577-X_27.

[4] **Ciardo, G.; Yu, A.** (2005): *Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning*. Correct Hardware Design and Verification Methods, 3725, pp. 146–161, DOI: 10.1007/11560548_13.

[5] **Clarke, E.; Grumberg, O.; Peled, D.A.** (1999): *Model Checking*. The MIT Press.

[6] **Emerson, E.A.; Clarke, E.M.** (1982): *Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons*. Sci. Comput. Program., 2(3), pp. 241–266.

[7] **Izquierdo-Rocha, J.; Sánchez-Perea, M.** (1994): *Application of the Integrated Safety Assessment methodology to the emergency procedures of a SGTR of a PWR*. Reliability Engineering and System Safety, 45, pp. 159–173, DOI: 10.1016/0951-8320(94)90083-3.

[8] **Jensen, K.; Kristensen, L.M.** (2009): *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, ISBN 978-3-642-00283-0.

[9] **Miner, A.** (2006): *Saturation for a General Class of Models*. Software Engineering, IEEE Transactions on, 32(8), pp. 559–570, DOI: 10.1109/TSE.2006.81.

[10] **Németh, E.; Bartha, T.** (2009): *Formal Verification of Safety Functions by Reinterpretation of Functional Block Based Specifications*. In: D. Cofer; A. Fantechi, eds., Formal Methods for Industrial Critical Systems, vol. 5596 of *Lecture Notes in Computer Science*, pp. 199–214, Springer Berlin / Heidelberg, ISBN 978-3-642-03239-4, DOI: 10.1007/978-3-642-03240-0_17.

[11] **Németh, E.; Bartha, T.; Fazekas, C.; Hangos, K.M.** (2009): *Verification of a primary-to-secondary leaking safety procedure in a nuclear power plant using coloured Petri nets*. Reliability Engineering and System Safety, 94 (5), pp. 942–953, DOI: 10.1016/j.ress.2008.10.012.

[12] **Tóth Heinemann, Z.** (2009): *Modelling and verification of discrete industrial control systems using formal methods*. Master's thesis, Budapest University of Technology and Economics (BME), [In Hungarian].

[13] **Vörös, A.; Darvas, D.; Bartha, T.** (2011): *Bounded Saturation Based CTL Model Checking*. In: J. Penjam, ed., Proc. of the 12th Symposium on Programming Languages and Software Tools, SPLST'11, pp. 149–160, Tallinn, Estonia, ISBN 978-9949-23-178-2.

[14] *Homepage of the PetriDotNet Framework*. URL: http://petridotnet.inf.mit.bme.hu/, last accessed May. 2012.

ACKNOWLEDGEMENT

This work was partially supported by the ARTEMIS JU and the Hungarian National Development Agency (NFÜ) in framework of the R3-COP project. Dániel Darvas and Attila Jámbor was partially supported by the MFB Hungarian Development Bank Plc. The authors would like to thank Prof. Gianfranco Ciardo for his valuable advice and suggestions.